Checking Model Transformation Refinement

Fabian Büttner[1], Marina Egea[2], Esther Guerra[3], and Juan de Lara[3]

[1] École des Mines de Nantes - INRIA (France)
fabian.buettner@inria.fr
[2] Atos Research & Innovation Dept., Madrid (Spain)
marina.egea@atosresearch.eu
[3] Universidad Autónoma de Madrid (Spain)
{Esther.Guerra, Juan.deLara}@uam.es

Abstract. Refinement is a central notion in computer science, meaning that some artefact S can be safely replaced by a refinement R, which preserves S's properties. Having available techniques and tools to check transformation refinement would enable (a) the reasoning on whether a transformation correctly implements some requirements, (b) whether a transformation implementation can be safely replaced by another one (e.g. when migrating from QVT-R to ATL), and (c) bring techniques from stepwise refinement for the engineering of model transformations.
In this paper, we propose an automated methodology and tool support to check transformation refinement. Our procedure admits heterogeneous specification (e.g. PAMOMO, Tracts, OCL) and implementation languages (e.g. ATL, QVT), relying on their translation to OCL as a common representation formalism and on the use of model finding tools.

1 Introduction

The raising complexity of languages, models and their associated transformations makes evident the need for engineering methods to develop model transformations [12]. Model transformations are software artefacts and, as such, should be developed using sound engineering principles. However, in current practice, transformations are normally directly encoded in some transformation language, with no explicit account for their *requirements*. These are of utmost importance, as they express *what* the transformation has to do, and can be used as a basis to assert correctness of transformation implementations. While many proposals for requirements gathering, representation and reasoning techniques have been proposed for general software engineering [15, 23], their use is still the exception when developing model transformations.

Specifications play an important role in software engineering, and can be used in the development of model transformations in several ways. First, they make explicit what the transformation should do, and can be used as a basis for implementation. Specifications do not necessarily need to be complete, but can document the main requirements and properties expected of a transformation. Then, they can be used as oracle functions for testing implementations [11].

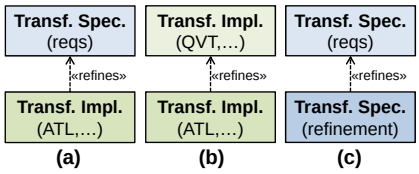


Fig. 1. Refinement scenarios.

In this setting, it is useful to know when a transformation T refines a specification S. Intuitively, this means that T can be used in place of S without breaking any assumption of the users of S. Some other times, we need to know whether a transformation T refines another transformation T' and can replace it. Fig. 1 gathers several scenarios where checking transformation refinement is useful. In (a), an implementation refines a requirements specification, hence ensuring correctness of the implementation with respect to the specification. In (b), a transformation implementation (e.g. in QVT) is refined by another one (e.g. in ATL) which can replace the former safely. This is especially useful if we want to migrate transformations, ensuring correctness of the migrated transformation. Finally, in (c), a specification refines another specification, which enables the application of stepwise refinement methodologies for transformation development.

In this paper, we tackle the previous scenarios by proposing an automated methodology to check transformation refinements. Our proposal relies on OCL as a common denominator for both specification languages (e.g. PAMOMO [13], Tracts [25] and OCL [18]) and transformation languages (e.g. QVT-R [21], triple graph grammars [22] and ATL [16]). For this purpose, we profit from previous works translating these languages into OCL [6, 7, 13, 25]. Hence, transformation specifications and implementations are transformed into *transformation models* [4] and we use SAT/model finding [6] techniques to automatically find counterexamples that satisfy properties assumed by the specification, but are incorrectly implemented. While refinement has been previously tackled in [25], our work is novel in that it proposes an automated procedure for performing this checking, and is able to tackle heterogeneous specification and transformation languages by using OCL as the underlying language for reasoning.

Paper organization Section 2 motivates the need for transformation refinement using an example. Section 3 introduces model transformation refinement. Section 4 details our methodology to check refinements. Section 5 provides more examples, Section 6 compares with related work and Section 7 concludes.

2 A Motivating Example

Assume we have gathered the requirements for the *Class2Relational* transformation, and want to use them as a blueprint to check whether an implementation correctly addresses them. Fig. 2 shows part of a specification of the requirements using the PAMOMO specification language [13], though we could choose any other transformation specification language instead (like Tracts or OCL).

PAMOMO is a formal, pattern-based, declarative, bidirectional specification language that can be used to describe correctness requirements of transformations and of their input and output models in an implementation-independent

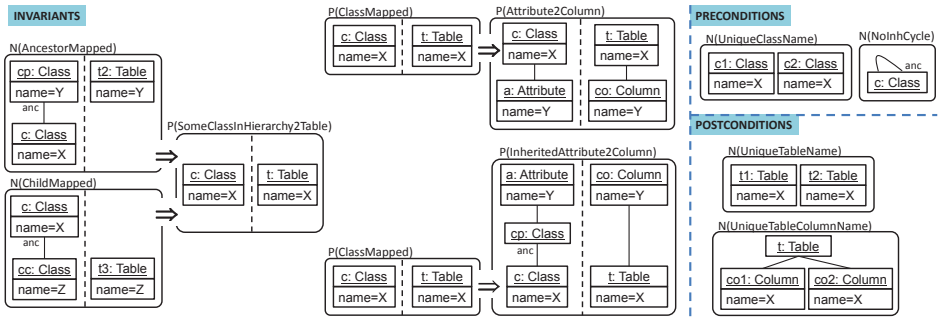


Fig. 2. A specification for the *Class2Relational* transformation.

way. These requirements may correspond to *pre-/postconditions* that input/output models should fulfill, as well as transformation *invariants* (i.e. requirements that the output model resulting from a particular input model should satisfy).

Preconditions, postconditions, and invariants are represented as graph patterns, which can be positive to specify expected model fragments, or negative to specify forbidden ones. They can have attached a logical formula stating extra conditions, typically (but not solely) constraining the attribute values in the graph pattern. Optionally, patterns can define one enabling condition and any number of disabling conditions, to reduce the scope of the pattern to the locations where the enabling condition is met, and the disabling conditions are not. The interested reader can find the formalization of these concepts in [13, 14].

Fig. 2 shows to the left three invariants that our *Class2Relational* transformation should fulfill. This specification is very general, in the sense that it gathers only the minimal requirements that any implementation of the *Class2Relational* should satisfy, leaving room for different transformation strategies. In particular, the specification only requires the transformation of at least one class in every class hierarchy, allowing freedom as to how many classes per hierarchy to transform (being 1 the minimum). This condition is checked by invariant *SomeClassInHierarchy2Table*, which states that if a class c does not have ancestors or children that have been transformed (disabling conditions *AncestorMapped* and *ChildMapped* respectively), then c should be transformed into a table (invariant *SomeClassInHierarchy2Table*). In the invariant, assigning the same variable to different attributes accounts for ensuring equality of their values, like X, which is assigned to the name of both the class c and the table t, meaning that both objects should have the same name. Moreover, relation *anc* is the transitive closure of the inheritance relation. Altogether, this invariant states that at least one class in each hierarchy should be transformed. The remaining invariants in this specification handle the correct transformation of attributes. To its right, invariant *Attribute2Column* states that if a class c is transformed (enabling condition *ClassMapped*), then its owned attributes should be converted into columns of the table. Below, invariant *InheritedAttribute2Column* states that all inherited attributes should be transformed into columns as well. Finally, another invari-

ant (omitted for reasons of space) states that attributes of non-mapped children classes should also be transformed for their mapped ancestors.

Fig. 2 shows to the right some pre- and postconditions for the input/output models. Whereas the shown invariants are positive and therefore their satisfaction is demanded, the shown pre- and postconditions are negative, indicating forbidden situations. Thus, precondition *UniqueClassName* forbids duplicated class names for input models, and *NoInhCycle* forbids having inheritance cycles. Three additional preconditions, not shown for space constraints, forbid duplicate attribute names in the same class, either defined locally or inherited. Similarly, postcondition *UniqueTableName* forbids duplicated table names in output models, and *UniqueTableColumnName* forbids two equally named columns in the same table. Note that although this is not the case, we could also define negative invariants, as well as positive pre- and postconditions, in specifications.

Developers can use the specification in Fig. 2 as a guide to implement the transformation in their favorite language. As an example, Listing 1 shows a possible implementation in ATL. The strategy followed is transforming each class and, in the generated table, creating columns coming from the attributes defined in the class or its ancestor classes (checked in line 10). The specification would also admit other transformation strategies, like mapping only top classes or only leaf classes. Note that the implementation transforms packages into schemas in lines 3–4, though the specification does not state how to handle them (there is just a multiplicity constraint saying that classes are always in a package).

```
1  module AllClasses; create OUT : SimpleRelational from IN : SimpleClass;
2
3  rule P2S { from p : SimpleClass!Package
4             to s : SimpleRelational!Schema }
5
6  rule C2T { from c : SimpleClass!Class
7             to t : SimpleRelational!Table ( name <- c.name, schema <- c.package ) }
8
9  rule A2A { from ua : SimpleClass!Attribute,
10                 c : SimpleClass!Class ( c = ua.owner or c.ancestors()->includes(ua.owner) )
11            to col : SimpleRelation!Column ( name <- ua.name, owner <- ua.owner ) }
```

Listing 1. Transformation implementation using ATL (*"AllClasses.atl"*).

Then, the question arises whether the implemented transformation is a refinement of the specification, i.e., whether for any valid input model (satisfying the preconditions), its transformation yields a model satisfying the invariants and postconditions of the specification. As we will see in Section 4, the answer to this question is *no* because this implementation does not guarantee the invariant *InheritedAttributeToColumn*, since the rule A2A contains a little bug (which we will uncover in Section 4.1). While finding this bug can be done manually by testing, in this paper we propose an automated procedure to detect the postconditions and invariants of the specification that are not satisfied.

Other scenarios for checking refinement are also of practical use. For example, if we want to migrate this transformation into QVT-R, we might want to ensure that the target transformation is compatible with the original one. The next section discusses the notion of *transformation refinement*, while Section 4 presents our approach to automatically assess the scenarios identified in Fig. 1.

3 Model Transformation Refinement

Conceptually, a model-to-model transformation $\mathcal{S}$ from a source metamodel $\mathcal{M}_{src}$ to a target metamodel $\mathcal{M}_{tar}$ can be represented by a relation $Sem(\mathcal{S})$ between pairs of source and target models of the metamodels[4].

$$Sem(\mathcal{S}) = \{(M_{src}, M_{tar}) : M_{src}\ \mathcal{S}\ M_{tar},\ \text{where}$$
$$M_{src} \text{ is a model of } \mathcal{M}_{src},\ \text{and } M_{tar} \text{ is a model of } \mathcal{M}_{tar}\}$$

A relation $\mathcal{S}$ does not need to be functional, i.e., the same source model may be related with several target models. In this way, we support both deterministic and non-deterministic transformations. Based on this characterization, we can express a refinement relation of a transformation.

Def. 1 (Refinement) *Given two transformation specifications $\mathcal{S}$, $\mathcal{S}'$ between a source metamodel $\mathcal{M}_{src}$, and a target metamodel $\mathcal{M}_{tar}$. $\mathcal{S}'$ refines $\mathcal{S}$ iff the following conditions hold:*

$$\forall M_{src}, M_{tar} : ((M_{src}, M_{tar}) \in Sem(\mathcal{S}') \ \wedge\ \exists M'_{tar} : (M_{src}, M'_{tar}) \in Sem(\mathcal{S})) \Rightarrow (M_{src}, M_{tar}) \in Sem(\mathcal{S}) \quad (1)$$

$$\forall M_{src}, M_{tar} : (M_{src}, M_{tar}) \in Sem(\mathcal{S}) \Rightarrow (\exists M'_{tar} : (M_{src}, M'_{tar}) \in Sem(\mathcal{S}')) \quad (2)$$

The second condition specifies the executability of $\mathcal{S}'$: $\mathcal{S}'$ must accept all inputs that $\mathcal{S}$ accepts. The first condition requires that S' behaves consistent to S on those inputs. Fig. 3 illustrates this relationship using a set notation. The source models accepted by $\mathcal{S}$ are given by set $Dom(\mathcal{S})$ (its domain, i.e., the models in the source metamodel of $\mathcal{S}$). The definition domain of $\mathcal{S}$ is the set $Ran(\mathcal{S})$ (its range, made of the models in the target metamodel of $\mathcal{S}$). Models are represented as dots, pairs of models in $Sem(\mathcal{S})$ are joined by a solid arrow, and pairs of models in $Sem(\mathcal{S}')$ are joined by dashed arrows. Fig. 3(a) shows a valid refinement as the upper pair in $Sem(\mathcal{S}')$ is also in $Sem(\mathcal{S})$. Since refinement is not concerned with source models not considered by $\mathcal{S}$, the lower source model is allowed to be related with any target model in $Sem(\mathcal{S}')$. Fig. 3(b) is not a refinement because the pair in $Sem(\mathcal{S}')$ is not in $Sem(\mathcal{S})$, while the source model of this pair is in $Dom(\mathcal{S})$. Fig. 3(c) is not a refinement, as $Sem(\mathcal{S}')$ misses one source model of $Dom(\mathcal{S})$. Altogether, the figure illustrates that the domain of $\mathcal{S}'$ should include the domain of $\mathcal{S}$ and be consistent with the elements in the domain of $\mathcal{S}$.

For the sake of simplicity, we assume that $\mathcal{S}$ and $\mathcal{S}'$ share the same metamodels. In practice, $\mathcal{S}$ may focus on the most important aspects of a transformation, while a refinement $\mathcal{S}'$ may be defined in more detail and over larger source/-target metamodels. Provided that the metamodels for $\mathcal{S}'$ are subtypes of the metamodels of $\mathcal{S}$ [24], we can always silently extent the metamodels for $\mathcal{S}$ to those of $\mathcal{S}'$ in Definition 1.

Some approaches, like model transformation contracts [9, 14], characterize the semantics of a transformation $\mathcal{S}$ from a source to a target metamodel by

[4] Notice that we are assuming source-to-target transformations.

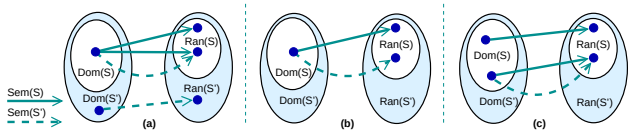


Fig. 3. Valid (a) and invalid (b, c) refinements.

means of three types of constraints: preconditions ($Pre_{\mathcal{S}}$), invariants ($Inv_{\mathcal{S}}$) and postconditions ($Post_{\mathcal{S}}$). We capture this semantics in Def. 2. In the definition, we represent by $Pre_{\mathcal{S}}$ the set of preconditions pre_i that the source models of $\mathcal{S}$ must fulfill, and use $Pre_{\mathcal{S}}(M_{src})$ to indicate that the model M_{src} fulfills all the preconditions of $\mathcal{S}$, i.e., $pre_1(M_{src}) \wedge \ldots \wedge pre_n(M_{src})$. We use a similar notation for invariants and postconditions as well. For this kind of transformations, assuming that their conditions can be translated into first-order logic, we can restate Def. 1 in terms of characterizing predicates as follows.

Def. 2 (Contract-based transformation specification) *Let $\mathcal{S}$ be a contract-based transformation specification from a source metamodel $\mathcal{M}_{src}$ to a target metamodel $\mathcal{M}_{tar}$. The relation set $Sem(\mathcal{S})$ defined by $\mathcal{S}$ can be characterized by three types of predicates that represent $\mathcal{S}$'s preconditions, invariants and postconditions (the contract of $\mathcal{S}$) in the following way*

$$Sem(\mathcal{S}) = \{(M_{src}, M_{tar}) : (M_{src} \in \mathcal{M}_{src}) \wedge (M_{tar} \in \mathcal{M}_{tar}) \wedge Pre_{\mathcal{S}}(M_{src}) \wedge Inv_{\mathcal{S}}(M_{src}, M_{tar}) \wedge Post_{\mathcal{S}}(M_{tar})\} \quad (3)$$

with the additional condition that $\forall(M_{src} \in \mathcal{M}_{src})$

$$(Pre_{\mathcal{S}}(M_{src}) \Rightarrow \exists(M_{tar} \in \mathcal{M}_{tar}) : Inv_{\mathcal{S}}(M_{src}, M_{tar}) \wedge Post_{\mathcal{S}}(M_{tar})) \quad (4)$$

Prop. 1 (Refinement for contract-based transformation specifications) *Let $\mathcal{S}$ and $\mathcal{S}'$ be contract-based transformation specifications from a source metamodel $\mathcal{M}_{src}$ to a target metamodel $\mathcal{M}_{tar}$. $\mathcal{S}'$ refines $\mathcal{S}$ iff the following conditions hold*

$$\forall(M_{src} \in \mathcal{M}_{src}, M_{tar} \in \mathcal{M}_{tar})(Pre_{\mathcal{S}}(M_{src}) \wedge Inv_{\mathcal{S}'}(M_{src}, M_{tar}) \wedge Post_{\mathcal{S}'}(M_{tar})) \Rightarrow (Inv_{\mathcal{S}}(M_{src}, M_{tar}) \wedge Post_{\mathcal{S}}(M_{tar})) \quad (5)$$

$$\forall(M_{src} \in \mathcal{M}_{src})\ (Pre_{\mathcal{S}}(M_{src}) \Rightarrow Pre_{\mathcal{S}'}(M_{src})) \quad (6)$$

Proof. We can show that using Def. 2, conditions (1) and (2) hold iff conditions (5) and (6) hold. The proof is included in the extended version of this paper[5].

This proposition allows checking refinement using satisfiability solving for transformations that can be characterized by contracts, as the next section will show.

[5] `http://www.emn.fr/z-info/atlanmod/index.php/ICMT_2013_Refinement`

Notice that Def. 2 characterizes an 'angelic' choice [3] for the executability: given a valid source model (w.r.t. $Pre_{\mathcal{S}}$), there must be at least one target model such that $Inv_{\mathcal{S}}$ and $Post_{\mathcal{S}}$ hold. We do not require that $Pre_{\mathcal{S}}$ and $Inv_{\mathcal{S}}$ always imply $Post_{\mathcal{S}}$, like one often expects an implementation to imply a postcondition in program verification. In our context, $Inv_{\mathcal{S}}$ is part of the specification, just as $Pre_{\mathcal{S}}$ and $Post_{\mathcal{S}}$.

Strong refinement. We demanded above that a refining transformation specification $\mathcal{S}'$ must accept all input models that the refined transformation S accepts (specified by $Pre_{\mathcal{S}}$), and that the output models of $\mathcal{S}'$ for those inputs are valid w.r.t. $Inv_{\mathcal{S}}$, and $Post_{\mathcal{S}}$. We did not characterize the effect of $\mathcal{S}'$ for input models not fulfilling $Pre_{\mathcal{S}}$. However, if we think of $Post_{\mathcal{S}}$ as a contract that *any* transformation execution needs to fulfill, it makes sense to define a new notion of refinement that we call *strong refinement*. Thus, $\mathcal{S}'$ is a strong refinement of $\mathcal{S}$ iff it is a refinement and $\forall(M_{src} \in \mathcal{M}_{src}, M_{tar} \in \mathcal{M}_{tar})$

$$((Pre_{\mathcal{S}'}(M_{src}) \wedge Inv_{\mathcal{S}'}(M_{src}, M_{tar}) \wedge Post_{\mathcal{S}'}(M_{tar})) \Rightarrow Post_{\mathcal{S}}(M_{tar})) \quad (7)$$

Previous works [25] have approached transformation refinement from a testing perspective. Hence, given a set of (manually created) input models, developers might discover an implementation result violating some postcondition or invariant, but cannot prove refinement. In the next section, we provide a stronger, automated methodology based on constraint solving to perform the checking.

4 Checking Refinement Using OCL Model Finders

Our methodology for checking transformation refinement builds on the fact that transformations in several declarative languages can be translated into a unified representation using OCL contracts. This unified representation, called *transformation model* [4], can be easily checked and analyzed using readily available OCL model finders. In short, the source and target metamodels are merged, and OCL constraints over this merged metamodel expresses the transformation semantics.

While such contracts are not directly executable, they are well-suited for automated checking of transformation properties as they allow expressing conditions covering the source and target models of the transformation at the same time. The checking can be done using a model finder, i.e., a satisfiability checker for metamodels, to verify the absence of counter examples for a given property. This way, for example, we have shown in [6] how to check if an ATL transformation can create output models that violate given constraints. Thus, we propose to generate and combine the OCL contracts for two transformation specifications in order to analyze the refinement relation between them following Prop. 1.

Fig. 4 shows the steps in our refinement checking methodology: (1) generation of the OCL contracts from both specifications $\mathcal{S}$ and $\mathcal{S}'$, (2) generation of counter example conditions, and (3) checking unsatisfiability of the counter-example conditions with an OCL model finder. Next, we detail these steps.

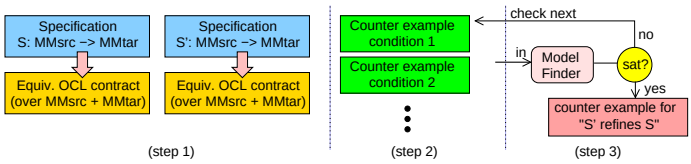


Fig. 4. Steps in the methodology to check refinement.

(1) Generation of OCL transformation contracts. First, each of the specifications $\mathcal{S}$ and $\mathcal{S}'$ is translated into equivalent sets of OCL constraints, $cons(\mathcal{S})$ and $cons(\mathcal{S}')$, over the combination of the source and target metamodels, $\mathcal{M}_{src}$ and $\mathcal{M}_{tar}$. Namely, $cons(\mathcal{S}) = precons(\mathcal{S}) \cup invcons(\mathcal{S}) \cup postcons(\mathcal{S})$, where $precons(\mathcal{S})$, $invcons(\mathcal{S})$, and $postcons(\mathcal{S})$ are OCL encodings of $Pre_{\mathcal{S}}$, $Inv_{\mathcal{S}}$, and $Post_{\mathcal{S}}$, fulfilling the conditions explained in Def. 2. Thus, in this setting we have $(M_{src}, M_{tar}) \in Sem(\mathcal{S})$ iff the evaluation of the constraints in $cons(\mathcal{S})$ over M_{src} combined with M_{tar} is true (and analogously for $\mathcal{S}'$).

Generators of such sets of OCL constraints have been described for several declarative, rule-based, specification/implementation transformation languages, including PAMOMO [13], QVT-R [7], triple grammars [7] and ATL [6].

(2) Generation of counter-example conditions. In order to check the two conditions for refinement of Prop. 1, we need that $invcons(\mathcal{S}) \wedge postcons(\mathcal{S})$ is implied by $precons(\mathcal{S}) \wedge invcons(\mathcal{S}') \wedge postcons(\mathcal{S}')$ and that $precons(\mathcal{S}')$ is implied by $precons(\mathcal{S})$ for every instance of $\mathcal{M}_{src}$ combined with $\mathcal{M}_{tar}$. This can be expressed as the following counter-example conditions that must all be unsatisfiable:

1. For each constraint c in $invcons(\mathcal{S}) \cup postcons(\mathcal{S})$, the set of constraints $precons(\mathcal{S}) \cup invcons(\mathcal{S}') \cup postcons(\mathcal{S}') \cup \{negated(c)\}$ must be unsatisfiable.
2. For each constraint c in $precons(\mathcal{S}')$, the set of constraints $precons(\mathcal{S}) \cup \{negated(c)\}$ must be unsatisfiable.
3. (For strong refinement) For each constraint c in $Post_{\mathcal{S}}$, the set of constraints $precons(\mathcal{S}') \cup invcons(\mathcal{S}') \cup postcons(\mathcal{S}') \cup \{negated(c)\}$ must be unsatisfiable.

If none of the counter-example conditions in 1–2 (1–3) is satisfiable, then $\mathcal{S}'$ refines (strongly refines) $\mathcal{S}$.

(3) Satisfiability checking of counter-example conditions. We use OCL model finders to check the counter-example conditions. There are several approaches for checking the satisfiability of OCL constraints, and our methodology is independent of them. For example, UML2Alloy [1] and the USE Validator [17] translate the problem into relational logic and use a SAT solver to check it, while UMLtoCSP [8] translates it into a constraint-logic program. The approach of Queralt et al. [20] uses resolution, and Clavel et. al [10] map a subset of OCL into a first-order logic and employ SMT solvers to check unsatisfiability. In this

paper, we have used the USE Validator because it supports a large subset of OCL and because the underlying SAT solver provides robust performance for a variety of problems. This tool performs model finding within given search bounds, using finite ranges for the number of objects, links and attribute values. Thus, when a counter example is found, we have proven that there is no refinement; if no counter example is found, we only know that the refinement is guaranteed up to the search bounds. However, not finding a counterexample is a strong indication of refinement if wide enough bounds are chosen for the search.

4.1 Running Example

In Sect. 2, we presented a specification of the *Class2Relational* transformation using PAMOMO (cf. Fig. 2), as well as a possible implementation of the *AllClasses* strategy using ATL (cf. Listing 1). Next, we illustrate our methodology by checking whether *AllClasses* refines *Class2Relational*.

(1) Generation of OCL transformation contracts. First, we generate the OCL contracts for *Class2Relational* and *AllClasses*. Following the compilation and tool support presented in [14], we generate one OCL invariant from each PAMOMO pattern. Listing 2 shows the OCL invariants for precondition *UniqueClassName*, invariant *Attribute2Column* and postcondition *UniqueTableColumnName*. These constraints belong to the sets *precons*(*Class2Relational*), *invcons*(*Class2Relational*) and *postcons*(*Class2Relational*), respectively. Notice that we silently assume a singleton class *GlobalContext* which hosts all OCL invariants. We refer the reader to [14] for a detailed presentation of this compilation scheme, and just highlight that the OCL expressions derived from preconditions only constrain the source models, those from postconditions only constrain the target models, and those from invariants constrain both.

```
1  context GlobalContext inv Pamomo_Pre_UniqueClassName:
2  not Class.allInstances()->exists(c1 | Class.allInstances()->exists(c2 | c2<>c1 and c1.name=c2.name))
3
4  context GlobalContext inv Pamomo_Inv_Attribute2Column:
5  Class.allInstances()->forAll(c |
6    Attribute.allInstances()->forAll(a | c.atts->includes(a) implies
7      Table.allInstances()->forAll(t | c.name=t.name implies
8        Column.allInstances()->exists(co | t.cols->includes(co) and a.name=co.name))))
9
10 context GlobalContext inv Pamomo_Pos_UniqueTableColumnName:
11 not Table.allInstances()->exists(t |
12   Column.allInstances()->exists(c1 | t.cols->includes(c1) and
13     Column.allInstances()->exists(c2 | c2 <> c1 and t.cols->includes(c2) and c1.name=c2.name)))
```

Listing 2. Some OCL invariants generated from the PAMOMO specification.

Then, we derive an OCL contract for the ATL implementation, following the rules and tool described in [6]. In this case, *precons*(*AllClasses*) and *postcons*(*AllClasses*) only contain the source and target metamodel integrity constraints, like multiplicity constraints. Listing 3 shows some OCL constraints in *invcons*(*AllClasses*). They control the matching of source objects, the creation of target objects and the bindings of properties in the target objects (see [6] for details).

```
1  context Attribute inv ATL_MATCH_A2A:
2    Attribute.allInstances()->forAll(I_ua |
3      Class.allInstances()->forAll(I_c | (I_c.ancestors()->includes(I_ua.owner)) implies
4          A2A.allInstances()->one(I_A2A | I_A2A.ua = I_ua and I_A2A.c = I_c)))
5
6  context A2A inv ATL_MATCH_A2A_COND: self.c.ancestors()->includes(self.ua.owner)
7
8  context C2T inv ATL_BIND_C2T_t_name: self.t.name=self.c.name
9  context A2A inv ATL_BIND_A2A_col_name: self.col.name=self.ua.name
10 context A2A inv ATL_BIND_A2A_col_owner: self.col.owner=self.c.c2t.t
11
12 context Column inv ATL_CREATE_Column: self.a2a->size()=1
```

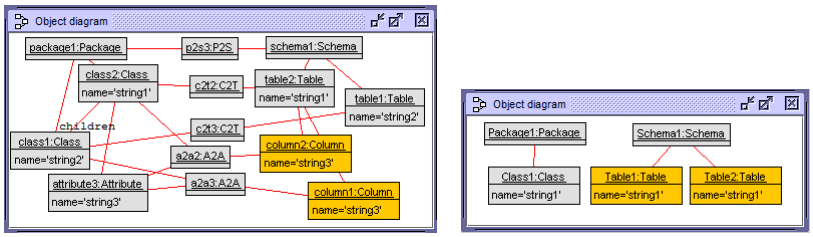
Listing 3. OCL invariants generated from the ATL rule *A2A*.

Notice that the mapping used for ATL [6] imposes a limitation for *invcons*(*AllClasses*): The OCL constraints use additional trace classes connecting the source and target objects in the transformation model. This means that we can use these constraints only in their positive form in the counter-example conditions, because for the negation we would need to express "there is no valid instance of the trace classes such that…", which is not available in OCL. In practice, this means that, using this OCL compilation, we can check whether an ATL specification refines any other transformation specification, but not the opposite. The compilations for PaMoMo and QVT-R do not have this limitation.

(2) Generation of counter-example conditions. From the full version of *Class2Relational* (Fig. 2 only shows an excerpt), we obtain 7 OCL invariants in *precons* (5 coming from PaMoMo preconditions and 2 from multiplicity constraints), 4 invariants in *postcons* (2 and 2), and 4 invariants in *invcons*. From the ATL version of *AllClasses*, we obtain 2 invariants in each *precons* and *postcons* for the multiplicity constraints, and 10 invariants in *invcons* characterizing the ATL rules. This gives 4 counter-example conditions to check for the first condition in Prop. 1, as explained on page 8, plus 2 cases for the second. If we want to check for strong refinement, we have 4 more counter-example conditions.

(3) Satisfiability checking of counter-example conditions. Checking the 10 counter-example conditions, for example with the USE Validator, yields the counter example shown in Fig. 5(a). The counter example satisfies all invariants that characterize *AllClasses* (hence it is a model of a valid ATL execution), but the OCL expression derived from the PaMoMo invariant *InheritedAttribute2Column* is violated (hence this pair of models is not in $Sem(Class2Relational)$). In particular, the problem is that the attribute inherited by *class1* is not attached to *table1*, but it is incorrectly attached to *table2*. Consequently, the instance is a counter example for postcondition *UniqueTableColumnName* as well.

If we examine the rule *A2A* in Listing 1 based on this counter example, we discover that the binding `owner<-ua.owner` is incorrect and should be changed to `owner<-c`. Fixing this error and checking the updated counter-example conditions again yields no counter example. Thus, the fixed version of *AllClasses* is a refinement of *Class2Relational*.



(a) *AllClasses* (ATL) (b) *BottomClasses* (checkonly)

Fig. 5. Refinement counter examples, checking against *Class2Relational*.

The ATL transformation is not a strong refinement of the PAMOMO specification though, since without demanding unique names in the source, ATL does not establish uniqueness of names in the target.

4.2 Tool support

For the first step in Fig. 4, the generation of transformation models, we have automated generators available for PAMOMO and ATL. So far, the generation from QVT-R is performed manually. For the second step, we have created a prototype to automate the construction of the counter-example conditions. For the third step, we call the USE Validator [17] to find refinement counter examples.

5 Further Examples

In this section, we discuss some more results for the case study. We have considered a 'zoo' of various specifications of *Class2Relational* using different languages (PAMOMO, ATL and QVT-R) and following three strategies (mapping all classes, only top classes, or only bottom classes). We have applied our methodology to check refinement for each pair of specifications (110 counter-example conditions in total). Fig. 6 shows the results. The absence of an arrow indicates no refinement (except for ATL, which can only be checked on the implementation side of the refinement relation). The details for all strategies are online[6], here we just highlight some interesting points.

We have considered two PAMOMO specifications: *Class2Relational* (the running example), and a refinement of this called *TopOrAll* which demands a 'uniform' mapping either of all classes in the source model, or only of the top ones.

The ATL implementation of the *TopClasses* strategy does not refine the *Class2Relational* specification. This strategy translates each top class into a table, and the attributes of the top class and its subclasses into columns of the table. However, if two subclasses of a top class have an attribute with the same name (which is not excluded by the preconditions of *Class2Relational*)

[6] `http://www.emn.fr/z-info/atlanmod/index.php/ICMT_2013_Refinement`

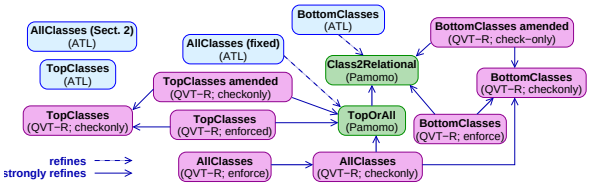


Fig. 6. Refinements between strategies (transitively reachable links are omitted).

then the generated table gets two columns with the same name, violating the postcondition *UniqueTableColumnName*. By contrast, the ATL versions of *AllClasses* (discussed in the previous section) and of *BottomClasses* are refinements of *Class2Relational*. *AllClasses* even refines the stronger specification *TopOrAll*.

Our methodology is also applicable to QVT-R. A QVT-R transformation S can be used in *enforce* mode to create a target model from scratch starting from a source model[7], or in *checkonly* mode to check the relation between an existing pair of models. Hence, we distinguish the sets $SEM_{ENF}(S)$ of source models and target models generated by S, and $SEM_{CHK}(S)$ of accepted pairs of source and target models. We will see that they are not equal.

Listing 4 shows the QVT-R implementation of the *BottomClasses* strategy. Interestingly, using this implementation in enforce mode is a refinement of the *Class2Relational* specification, but using it in checkonly mode is not. This is because the checkonly mode checks for the elements that should be created by the enforce mode, but the target model can contain more elements. The problem is that these extra elements can violate invariants or postconditions from the requirements specification. For example, Fig. 5(b) shows a refinement counter example violating the postcondition *UniqueClassName*, while satisfying the QVT-R transformation in checkonly mode.

```
1  transformation BottomClasses (source : SimpleClass, target : SimpleRelational) {
2    key SimpleRelational::Table {name};
3    key SimpleRelational::Column {owner, name};
4
5    top relation PackageToSchema {
6      checkonly domain source p : SimpleClass::Package {};
7      enforce domain target s : SimpleRelational::Schema {}; }
8
9    top relation ClassToTable {
10     cn : String;
11     checkonly domain source c : SimpleClass::Class {
12       package = p : SimpleClass::Package {}, name = cn };
13     enforce domain target t : SimpleRelational::Table {
14       schema = s : SimpleRelational::Schema {}, name = cn };
15     when { c.children−>size()=0 and PackageToSchema(p, s); }
16     where { AttributeToColumn(c, t); SuperAttributeToColumn(c, t); } }
17
18   relation AttributeToColumn {
19     an : String;
20     checkonly domain source c : SimpleClass::Class {
21       atts = a : SimpleClass::Attribute { name = an } };
22     enforce domain target t : SimpleRelational::Table {
23       cols = cl : SimpleRelational::Column { name = an } }; }
```

[7] QVT-R also supports the incremental scenario, but we leave it out here.

```
24
25   relation SuperAttributeToColumn {
26     checkonly domain source c : SimpleClass::Class {
27       package = p : SimpleClass::Package { classes = sc : SimpleClass::Class {} } };
28     enforce domain target t : SimpleRelational::Table {};
29     when { c.ancestors()->includes(sc); }
30     where { AttributeToColumn(sc, t); } }
31 }
```

Listing 4. QVT-R implementation of the *BottomClasses* strategy.

In order to make the checkonly transformation a refinement of *Class2Relational*, we need to include a top-level relation stating that non-bottom classes do not have an associated table. This extra relation is non-constructive and is not concerned with the creation of target elements, but with their absence. As ATL can only be used in enforce mode, this constraint is built-in into ATL.

Finally, the 'check-before-enforce' semantics of QVT-R prevents the creation of new objects if equivalent ones exist in the target. The equivalence criteria for objects are given through keys. By setting an appropriate key for columns (see line 3 in Listing 4) we avoid having repeated columns in tables. This is why the enforce mode of the QVT-R transformation for the *TopClasses* strategy correctly refines the *ClassToRelational* specification (whereas the ATL implementation does not, as explained above).

Regarding performance, for the examples considered so far, solving times using the USE Validator have not been an issue (within a few seconds for a default search bound of 0..5 objects per class). It remains as future work to evaluate the scalability on larger examples.

6 Related Work

To our knowledge, the only work addressing transformation refinement is [25]. Its authors use Tracts to build transformation contracts. Tracts are OCL invariants that can be used to specify preconditions, postconditions and transformation invariants. The authors introduce the following notion of refinement: a Tract $\mathcal{S}'$ refines another one $\mathcal{S}$ if $\mathcal{S}'$ has weaker preconditions, but stronger invariants and postconditions $((Pre_{\mathcal{S}} \Rightarrow Pre_{\mathcal{S}'}) \wedge (Inv_{\mathcal{S}'} \Rightarrow Inv_{\mathcal{S}}) \wedge (Post_{\mathcal{S}'} \Rightarrow Post_{\mathcal{S}}))$. This is a safe approximation to replaceability as in Def. 1, while our Prop. 1 exactly characterizes this notion. Moreover, we also distinguish *strong refinement*.

Regarding refinement checking, in [25], refinement is checked by building a suitable set of input test models and testing S' against S's pre/postconditions and invariants. This approach has two drawbacks. First, it is based on testing and on the manual creation of input test models. Secondly, it assumes that S' is an executable implementation which can be used for testing. As we have seen in this paper, S' might be a non-executable specification.

Our checking procedure ensures correctness criteria for the refining transformation. In this respect, the work in [19] provides a means to verify a transformation against verification properties, assuming that both are given by patterns, in the line of PAMOMO patterns. Verification properties are restricted to be positive. The checking implies generating all minimal glueings of the transformation

patterns, and checking them against the verification property. In such restricted case, the verification is finitely terminating. We plan to investigate the glueing minimality conditions to provide suitable search bounds for the solver.

The use of OCL to define transformation contracts was proposed in [9]. This idea was extended in [4] with the aim to build transformation models as a declarative means to capture the transformation semantics. Transformation models with OCL constraints were used for transformation verification using model finders in [2, 6, 7]. None of these works propose checking transformation refinement.

7 Conclusions and Future Work

In this paper, we have presented a methodology and tool support to check transformation refinement. Refinement is useful to check whether an implementation is correct with respect to a specification, to ensure replaceability of implementations (e.g. when migrating a transformation), and to apply step-wise refinement techniques to transformation development.

Our methodology can be applied to check refinement between transformations in any specification or implementation language for which a translation to an OCL transformation model exists. To our knowledge, such translations exist for PAMOMO, QVT-R, TGGs, and for a subset of ATL without imperative code blocks. One limitation of the OCL contract-based approach is that recursive rules cannot be generally mapped into OCL contracts, since OCL has no fix-point operator. For example, the QVT-R-to-OCL translation in [7] would, for recursive rules, yield recursive helper operations. For bounded verification, however, such definitions can be still statically unfolded up to a given depth.

Our methodology is actually independent of OCL. For example, it would apply to transformations contracts specified in first order logic, too, like in [5]. That would open up further possibilities for symbolic reasoning. Our lightweight methodology permits checking transformation correctness; however, as it relies on bounded model finding, formally, our method can only disprove refinement. Using wide enough bounds can provide high confidence in refinement, though. Another possibility would be to prove implications from the invariants and post-conditions of S' to those of S; however, this would require the use of theorem provers, with less automation. We will explore this path in future work. Finally, we also plan to combine the constraints coming from the implementation and the specification to derive models for testing, in the style of [11].

Acknowledgements. Research partially funded by the Nouvelles Équipes Program of the Pays de la Loire Region (France), the EU project NESSoS (FP7 256890), the Spanish Ministry of Economy and Competitivity (project "Go Lite" TIN2011-24139), the R&D programme of the Madrid Region (project "e-Madrid" S2009/TIC-1650).

References

1. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.

2. K. Anastasakis, B. Bordbar, and J. Küster. Analysis of model transformations via alloy. In *MODEVVA'07*, 2007.
3. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 1998.
4. J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? Transformation models! In *MoDELS'06*, volume 4199 of *LNCS*. Springer, 2006.
5. F. Büttner, M. Egea, and J. Cabot. On verifying ATL transformations using off-the-shelf SMT solvers. In *MoDELS'12*, volume 7590 of *LNCS*, pages 432–448. Springer, 2012.
6. F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL transformations using transformation models and model finders. In *ICFEM'12*, volume 7635 of *LNCS*, pages 198–213. Springer, 2012.
7. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
8. J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE'07*. ACM, 2007.
9. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the specification of model transformation contracts. In *OCL Workshop*, volume 12, pages 69–83, 2004.
10. M. Clavel, M. Egea, and M. A. G. de Dios. Checking Unsatisfiability for OCL Constraints. *Electronic Communications of the EASST*, 24:1–13, 2009.
11. E. Guerra. Specification-driven test generation for model transformations. In *ICMT'12*, volume 7307 of *LNCS*, pages 40–55. Springer, 2012.
12. E. Guerra, J. de Lara, D. Kolovos, R. Paige, and O. dos Santos. Engineering model transformations with transML. *Software and Systems Modeling*, in press, 2012.
13. E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige. A visual specification language for model-to-model transformations. In *VL/HCC'10*, pages 119–126. IEEE CS, 2010.
14. E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46, 2013.
15. D. Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT, 2012.
16. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comp. Pr.*, 72(1-2):31–39, 2008.
17. M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *TOOLS'11*, volume 6705 of *LNCS*, pages 290–306. Springer, 2011.
18. OMG. *OCL Specification, version 2.3.1 (Document formal/2012-01-01)*, 2012.
19. F. Orejas and M. Wirsing. On the specification and verification of model transformations. volume 5700 of *LNCS*, pages 140–161. Springer, 2009.
20. A. Queralt and E. Teniente. Verification and validation of UML conceptual schemas with OCL constraints. *TOSEM*, 21(2):13, 2012.
21. QVT. `http://www.omg.org/spec/QVT/1.0/PDF/`. Last accessed: Nov. 2010, 2005.
22. A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
23. J. M. Spivey. An introduction to Z and formal specifications. *Softw. Eng. J.*, 4(1):40–50, 1989.
24. J. Steel and J.-M. Jézéquel. On model typing. *SoSyM*, 6(4):401–413, 2007.
25. A. Vallecillo and M. Gogolla. Typing model transformations using Tracts. In *ICMT'12*, volume 7307 of *LNCS*, pages 56–71. Springer, 2012.